C12

```
/*
** ==================================================================
                                                            ========
** Copyright 1996,1997 EMC Corporation
** ==================================================================
                                                            ========
*/
/*
** ==================================================================
                                                            ========
** DDRSTsvc_init.c
**
** Mission Statement:
**
**
**
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc
                                                support.  If
**                              not set, assume DCE support.
**
** Basic idea here:
** ==================================================================
                                                            ========
*/
/*
** ==================================================================
                                                            ========
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
** ==================================================================
                                                            ========
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMccr.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <string.h>
#include <stdlib.h>
#include <pthread.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>
```

```
#include <csc/cscomm.h>
#include <edmlink/edmlink_api.h>

#ifdef __cplusplus
extern "C" {
#endif

#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol.h>
#include <restore/RestoreObjectID.h>
#include <restore/csc_Dispatch_Protocol_Service.h>
#include <restore/dispatch_protocol_service.h>
#include <restore/dispatch_protocol_client.h>
#include <dpService.h>

#ifdef __cplusplus
}
#endif

#include <logging/logging.h>
#include <EDMDispatchLog.h>
#include <EDMDDHandle.h>
#include <EDMDDHandleMgrApi.h>
#include <EDMSession.h>
#include <EDMccr.h>
#include <EDMutils.h>
#include <EDMDD_ddp.h>
#include <EDMDDcr_rstsvc.h>

pthread_cond_t cscPortRdy_cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t cscPortRdy_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t G_serviceMtx;

static boolean32 print_error = TRUE;

/* Prototypes */
int edmrst_send_chndl_to_private_svc(int);
int edmrst_create_ddp_client_connection(
                        int,rpc_binding_handle_t **,EDMSession  *);
int edmrst_send_uid_to_private_svc(int, EDMSession *);

/* Dispatch Protocol ifspec */
static rpc_if_handle_t DispatchDaemon_ifspec;
ELinkHandlePtr_ty  ELinkHandle;     /* Handle for svc object */

/*******************************************************************
                                                            ********
**
** Routine:  LockSvcMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the mutex for the service execution
**
********************************************************************
                                                            ********
*/

static void
LockSvcMutex()
```

```
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_serviceMtx, NULL);
    }

    pthread_mutex_lock(&G_serviceMtx);
}
```

```
/***************************************************************************
                                                                   ********
**
** Routine:  UnlockSvcMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for service execution
**
****************************************************************************
                                                                   ********
*/

static void
UnlockSvcMutex()
{
    pthread_mutex_unlock(&G_serviceMtx);
}
```

```
void *
DDRSTsvc_init(void *pSessObj)
{
int                         lrc;
                                    /* Local Return Code              */
int                         fd1;
int                         fd2;
int                         status;
rpc_binding_handle_t *  bh=NULL;
EDMSession *                p_so;
// EDMDDHandle *                    pEHandleObj;
DD_client_session_id    sID;
ELinkShellObjPtr_ty     ShellHandle;
unsigned char           *svc_rpc_h=NULL; /* X-Service RPC Handle */
ELinkTargetObjPtr_ty    TargetObjPtr;
                                    /* Target object all functions    */
ELinkUserIdObjPtr_ty    UserIdObjPtr;
                                    /* UserId object copy & shell     */
ELinkCmdObjPtr_ty       CmdObjPtr;
                                    /* Shell command shell only       */
unsigned long           options = 0;
                                    /* For ELinkNewServiceLaunchObj   */


   //
   // launch one service at a time.
   //
   LockSvcMutex();
   pthread_mutex_lock( &cscPortRdy_mutex );

   //
   // Check to see that the EDMLINK handle didn't get trashed
   //
   if (ELinkHandle == NULL)
   {
       UnlockSvcMutex();
       pthread_mutex_unlock( &cscPortRdy_mutex);
       pthread_exit( NULL );
   }

   //
   // Cast the input argument to its object type.
   //
   p_so = (EDMSession*)pSessObj;

   //
   // Construct EDM-Link target object so that EDM-Link will know what
   // system we want to talk to.
   //

   TargetObjPtr = ELinkNewTargetObj( ELinkHandle,
                                     "localhost" );

   //
   // EDM-Link should have called our callback DOMIELinkCallback which
   // should have loaded DOMIHandle->ErrorBlock, so all we have to do
   // now, is return.
   //
   if ( NULL == TargetObjPtr )
   {
       p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
       UnlockSvcMutex();
       pthread_mutex_unlock( &cscPortRdy_mutex);
       pthread_exit( NULL );
   }
```

```
   //
   // Construct EDM-Link user object.
                                                We always want to run as root on the
   // target.
                    We are starting a private service that will run on an EDM.
   // We know that we will be starting via the EDM-Link daemon and we
                                                                              can
   // always start using the root id.  Also, this will be a service,
                                                                         it needs
   // to run as root and will have some intelligence in protecting
                                                                        itself in
   // that there a limited things that it can do and the caller of the
                                                                              API
   // will control what can be done.
   //
   UserIdObjPtr = ELinkNewUserIdObj( ELinkHandle,
                                     TargetObjPtr,
                                     NULL,
                                     NULL );

   //
   // EDM-Link should have called our callback DOMIELinkCallback which
   // should have loaded DOMIHandle->ErrorBlock, so all we have to do
   // now, is return.
   //
   if ( NULL == UserIdObjPtr )
   {
       (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
       p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
       UnlockSvcMutex();
       pthread_mutex_unlock( &cscPortRdy_mutex);
       pthread_exit( NULL );
   }

   //
   // Utilize the EDM-Link service launcher to physically startup the
   // domain private service.  By convention, all private services can
   // be found in /usr/epoch/service and have a suffix of pd.
                                                                       The domain
   // private service is: /usr/epoch/service/domainpd.
   //

   if (IsDebugOn())
       options |= ELINK_SERVICE_DEBUG;
                                       /* if we are debug, so will be R.Eng */

   CmdObjPtr = ELinkNewServiceLaunchObj( ELinkHandle,
                                         TargetObjPtr,
                                         "edmrestoreeng",
                                           /* Domain private service */
                                         options );
   //
   // EDM-Link should have called our callback DOMIELinkCallback which
   // should have loaded DOMIHandle->ErrorBlock, so all we have to do
   // now, is return.
   //
   if ( NULL == CmdObjPtr )
   {
       (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
       (void) ELinkDestroyObj( ELinkHandle, UserIdObjPtr );
       p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
       UnlockSvcMutex();
       pthread_mutex_unlock( &cscPortRdy_mutex);
       pthread_exit( NULL );
   }
```

```
  //
  // Fire up Private Service via EDM-Link API ELinkPrivateSvc.  This
  // physically starts the private service running.
  //
  lrc = ELinkPrivateSvc ( ELinkHandle,
                          TargetObjPtr,
                          UserIdObjPtr,
                          CmdObjPtr,
                          &fd1,
                          &fd2,
                          &ShellHandle );
  if ( -1 == lrc )
    {
      (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
      (void) ELinkDestroyObj( ELinkHandle, UserIdObjPtr );
      (void) ELinkDestroyObj( ELinkHandle, CmdObjPtr );
      EDMDispatch_logent(
                __FILE__,__LINE__,LOG_ERR,DDP_PRIVATE_SVC_FAILURE,
                        0,"ELinkPrivateSvc() failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_EXEC);
      UnlockSvcMutex();
      pthread_mutex_unlock( &cscPortRdy_mutex);
      pthread_exit( NULL );
    }

  (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
  (void) ELinkDestroyObj( ELinkHandle, UserIdObjPtr );
  (void) ELinkDestroyObj( ELinkHandle, CmdObjPtr );

  //
  // Extract the csc handle from the shell object. This handle
  // is the restore service (restore API) rpc handle.
  //
  svc_rpc_h = (unsigned char*) calloc(1,CONNECT_HANDLE_SIZE);
  if (svc_rpc_h == NULL)
  {
      EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_NO_MEMORY,
                         0,"calloc() failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_mutex_unlock( &cscPortRdy_mutex);
      pthread_exit( NULL );
  }
  lrc = ELinkGetConnectHandle( ELinkHandle,
                               ShellHandle,
                              svc_rpc_h );
  if ( 0 != lrc )
    {
      (void) free(svc_rpc_h);
      EDMDispatch_logent(
           __FILE__,__LINE__,LOG_ERR,DDP_GET_CONNECT_HANDLE_FAILURE,
                 0,"edmrst_get_client_rpc_handle() failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_mutex_unlock( &cscPortRdy_mutex);
      pthread_exit( NULL );
    }

  p_so -> setConnectionHandle((void *)svc_rpc_h);

  p_so -> getSessionID(&sID);           // Get Unique Session id

  //
  // Issue message telling of Dispatch Daemon RDR port number.
  //
```

```
  if (IsDebugOn())
  {
      EDMDispatch_logent(__FILE__,__LINE__,LOG_INFO,DDP_PORT_NUMBERS,
                         0,"PORT_INFO DispatchDaemon_ifspec(
                                                  DDCCR) port#: %d",
                         DispatchDaemon_ifspec.portnum);
  }

  //
  // Unlock Port Rdy mutex so the Reader can listen.
  //
  pthread_mutex_unlock( &cscPortRdy_mutex);

  //
  // Tell the Dispatch Daemon Protocol Reader Thread to Listen.
  //
  pthread_cond_signal(&cscPortRdy_cv);

  //
  // Inform the restore svc of dispatch protocol details (
                                                       port etc ...)
  //
  lrc = edmrst_send_chndl_to_private_svc(fd1);
  if ( 0 != lrc )
  {
      (void) free(svc_rpc_h);
      EDMDispatch_logent(
                __FILE__,__LINE__,LOG_ERR,DDP_CHANNEL_SEND_FAILURE,
                        0,"edmrst_send_chndl_to_private_svc(
                                                        ) failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_exit( NULL );
  }

  //
  // Send the Unique Session Id value.
  //
  lrc = edmrst_send_uid_to_private_svc(fd1,p_so);
  if ( 0 != lrc )
  {
      (void) free(svc_rpc_h);
      EDMDispatch_logent(
                 __FILE__,__LINE__,LOG_ERR,DDP_SEND_UID_FAILURE,
                        0,"edmrst_send_uid_to_private_svc(
                                                        ) failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_exit( NULL );
  }

  //
  // Create the CCW service handle so we can respond to messages.
  //
  lrc = edmrst_create_ddp_client_connection(fd1, &bh, p_so);
  if ( 0 != lrc )
  {
      (void) free(svc_rpc_h);
      EDMDispatch_logent(
            __FILE__,__LINE__,LOG_ERR,DDP_CREATE_CLIENT_CONNECTION,
                        0,"edmrst_create_ddp_client_connection(
                                                        ) failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_exit( NULL );
  }

  //
  // Insert handle object into Global list.
  //
  lrc = newHandleSet( &sID,
                      fd1,
                      fd2,
                      bh,
                     &ShellHandle,
                      &status );
  if ( 0 != lrc )
  {
     (void) free(svc_rpc_h);
     EDMDispatch_logent(
          __FILE__,__LINE__,LOG_ERR,DDP_HANDLE_INSERTION_ERROR,
                    status,"newHandleSet() failure");
     p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
     UnlockSvcMutex();
     pthread_exit( NULL );
  }

  //
  // Let's clean up and set the status to RUNNING.
  //
  p_so -> setStatus(DD_SERVICE_RUNNING);
  UnlockSvcMutex();
  pthread_exit( NULL );
 return( NULL );
}
```

```
/*
** ==================================================================
                                                            ========
**  Function:    edmrst_send_chndl_to_private_svc()
**  Description:
**
**
**
**  Returns:      0 Sucessful
**               -1 Read Failure
**               <0 Read less than exspected
** ==================================================================
                                                            ========
*/
int
edmrst_send_chndl_to_private_svc(int pipeToSvc)
{
auto int lrc=0;
auto unsigned char *p_client_h=NULL;

//
//  Isolate the connection handle from the server 'if_spec'.
//  The IP/PORT are part of the created if_spec structure.
//
p_client_h = DispatchDaemon_ifspec.connect_handle_p;

//
// Write the handle to the service so it can contact me
//
lrc = edmrst_WrChannel(pipeToSvc,
                       p_client_h,
                       CONNECT_HANDLE_SIZE);
if ( CONNECT_HANDLE_SIZE != lrc )
  {
    (void) free(p_client_h);
    EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_WRITE_CHANNEL,
                       0,"edmrst_WrChannel() Failure");
    return(-1);
  }

return(0);
}
```

```
/*
** ====================================================================
                                                              ========
**  Function:    edmrst_send_uid_to_private_svc()
**  Description:
**
**
**
**  Returns:      0 Sucessful
**               -1 Read Failure
**               <0 Read less than exspected
** ====================================================================
                                                              ========
*/
int
edmrst_send_uid_to_private_svc(int pipeToSvc,
                               EDMSession  *pSessionObj)
{
auto int lrc=0;
auto DD_client_session_id uid;

//
// Write the handle to the service so it can contact me
//
pSessionObj -> getSessionID(&uid);
lrc = edmrst_WrChannel(pipeToSvc,
                       (void*)&uid,
                       sizeof(DD_client_session_id));
if ( (sizeof(DD_client_session_id)) != lrc )
  {
    EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_WRITE_CHANNEL,
                       0,"edmrst_WrChannel() Failure");
    return(-1);
  }

return(0);
}
```

```
/*
** ====================================================================
                                                              ========
**  Function:    edmrst_create_ddp_client_connection()
**  Description:
**
**
**
**  Returns:      0 Sucessful
**               -1 Read Failure
**               <0 Read less than exspected
** ====================================================================
                                                              ========
*/
int
edmrst_create_ddp_client_connection(int pipeToSvc,
                                    rpc_binding_handle_t **bh,
                                    EDMSession  *p_so )
{
 int lrc;
 unsigned char *p_restore_service=NULL;
 error_status_t status;
 rpc_if_handle_t *p_psvc_ifspec=NULL;
 rpc_binding_handle_t *psvc_h=NULL;

 //
 // We now need to get the details from the restore service on
 // how to connect from the dispactch daemon ccw to the restore
 // service ccr. At this point, the restore service will be send -
 // ing the restore service ccr handle information. The port / ip
 // are the key information needed to create the ddp ccw handle.
 //
 lrc = edmrst_get_client_handle( pipeToSvc,&p_restore_service );
 if ( 0 != lrc )
 {
   EDMDispatch_logent(
                 __FILE__,__LINE__,LOG_ERR,DDP_GET_CLIENT_HANDLE,
                    0,"edmrst_get_client_handle() failure");
   p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
   return(-1);
 }

 //
 // Create an ifspec from the handle
 //
 p_psvc_ifspec = (rpc_if_handle_t *)
              calloc(1,sizeof(rpc_if_handle_t ));
 if (p_psvc_ifspec == NULL)
 {
  EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_NO_MEMORY,
                     0,"ifspec calloc() failure");
  return(-1);
 }

 lrc = csc_private_ifspec_init( p_restore_service,
                                EDM_DISPATCH_PROTOCOL_CLIENT,
                                EDMDPC_FUNCTIONS,
                                p_psvc_ifspec,
                                &status );
 if ( 1 != lrc )
 {
   (void) free(p_psvc_ifspec);
   EDMDispatch_logent(
                 __FILE__,__LINE__,LOG_ERR,DDP_IFSPEC_INIT_FAILURE,
                   status,"csc_private_ifspec_init() Failure");
   return(-1);
   }

   if (IsDebugOn())
   {
       EDMDispatch_logent(__FILE__,__LINE__,LOG_INFO,DDP_PORT_NUMBERS,
                       0,"PORT_INFO p_psvc_ifspec(DDCCW) port#: %d",
                       p_psvc_ifspec->portnum);
   }

   psvc_h = (rpc_binding_handle_t *) calloc(1, sizeof(
                                           rpc_binding_handle_t));
   //
   // Using the connect handle (128 bytes) received from the restore
   // service, connect to the restore service.
   //
   lrc = csc_connect_to_async_rpc_service( NULL,
                                           *p_psvc_ifspec,
                                           psvc_h,
                                           &status );
   if ( 1 != lrc )
   {
     (void) free(p_psvc_ifspec);
     (void) free(psvc_h);
     EDMDispatch_logent(
             __FILE__,__LINE__,LOG_ERR,DDP_PRIVATE_SVC_CONNECT_FAILURE,
                         status,"csc_connect_to_async_rpc_service(
                                ) Failure. Status is %d", status);
     return(-1);
   }

  *bh = psvc_h;
  (void) free(p_psvc_ifspec);

  return(0);
}
```

```
/*
** ==========================================================================
                                                                  ========
**  Function:
**  Description:
**
**
**
**  Returns:      0 Sucessful
**               -1 Read Failure
** ==========================================================================
                                                                  ========
*/
int
EDMDDSvcInit()
{
struct hostent         *hp;
struct utsname         name;
error_status_t         csc_status;
  int lrc = 0;

    ELinkHandle = ELinkInitAPI(ELINK_SHELL_EDMLINK);

    if (ELinkHandle == NULL)
    {
        return -1;
    }

   //
   // Initialize the ifspec specification from the private svc
   // creation call. This call will output the DispatchDaemon_ifspec
   //
    lrc = csc_async_ifspec_init (&DispatchDaemon_ifspec,
                                CSC_IFSPEC_PRIVATE_TYPE,
                                DP_PROGNUM,
                                DP_VERSNUM,
                                (
              dispatch_func_p_t)&edm_dispatch_protocol_service_1_table,
                                &csc_status);


   if ( TRUE != lrc )
   {
       EDMDispatch_logent(
                       __FILE__,__LINE__,LOG_ERR,DDP_IFSPEC_INIT_FAILURE,
                           csc_status,"csc_async_ifspec_init() Failure");
       return(-1);
   }


   //
   //  We need the system name and ip for the if_spec.
   //
   uname( &name );
   hp = gethostbyname( name.nodename );
   if ( NULL == hp )
    {
        EDMDispatch_logent(
                       __FILE__,__LINE__,LOG_ERR,DDP_GETHOSTNAME_FAILURE,
                                0,"gethostbyname() failure");
        return -1;
    }

   ( void ) memcpy( (char*) &DispatchDaemon_ifspec.ip_addr,
                    hp->h_addr, hp->h_length );
  //
  // Register the callback functions.
  //
  lrc = csc_register_async_server_interface(
                              &DispatchDaemon_ifspec,
                              -1,
                              edm_dispatch_protocol_service_1_table,
                              edm_dispatch_protocol_service_1_nproc,
                              &csc_status );

  if ( TRUE != lrc )
  {
    EDMDispatch_logent(
               __FILE__,__LINE__,LOG_ERR,DDP_REGISTER_SVC_FAILURE,
                   csc_status,
               "Failed to register asynchronous server interface.");
    return -1;
  }

   return 0;
}
```

```
/*
** ==================================================================
                                                            ========
** Copyright 1996,1997 EMC Corporation
** ==================================================================
                                                            ========
*/
/*
** ==================================================================
                                                            ========
** EDMDD_ccw.c
**
** Mission Statement: This is the entry point for the Control Channel
                                                      Writer thread.
**                   Its main purpose is to write notifications or
                                                     progress to the
**                   Dispatch Daemon.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                        USE_SUNRPC - Compile source with sunrpc
                                                    support.   If
                                  not set, assume DCE support.
**
**
** Basic idea here: Module for Control Channel Writer thread.
** ==================================================================
                                                            ========
*/
/*
** ==================================================================
                                                            ========
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
** ==================================================================
                                                            ========
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMccw.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <pthread.h>
#include <csc/cscomm.h>

#include <rw/collect.h>

#ifdef __cplusplus
extern "C" {
#endif

#include <restore/dispatch_protocol.h>
#include <restore/dispatch_protocol_client.h>
#include <restore/csc_Dispatch_Protocol_Client.h>

#ifdef __cplusplus
}
```

```
#endif

#include <EDMDD_ccw.h>
#include <EDMDD_ddp.h>
#include <EDMReturnMessageApi.h>
#include <EDMccw.h>
#include <EDMutils.h>
#include <EDMDDHandleMgrApi.h>
#include <EDMDispatchSession.h>
#include <logging/logging.h>
#include <EDMDispatchLog.h>

extern ELinkHandlePtr_ty  ELinkHandle;

// Internal Routines
static int SendConnectConfirmMessage(
                        DD_client_session_id*,rpc_binding_handle_t*);
static int SendAbortRequestMessage(
                        DD_client_session_id*,rpc_binding_handle_t*);
static int SendCloseRequestMessage(
                        DD_client_session_id*,rpc_binding_handle_t*);
static int SendPingRequestMessage(
                        DD_client_session_id*,rpc_binding_handle_t*);
static int SendFinalStatsConfirmMessage(
                        DD_client_session_id*,rpc_binding_handle_t*);

void *
DispDaemon_ccw(void *buf)
{
int rc=0;
int ResponseMessage=0;
int status=0;
int sstatus=0;
DD_client_session_id sid;
rpc_binding_handle_t *client_h_p=NULL;

for( ; ; )
  {
    /* Monitor the event queue for messages to send. */
    rc = PopResponseMessage(&ResponseMessage,
                            &sid,
                            &client_h_p,
                            &status);
    if ( -1 == rc )
    {
        sleep( DISPATCH_CCW_SLEEP );
         continue;
    }

     rc = GetSessionStatus(&sid, &sstatus, &status);

     if ((rc != 0) && (ResponseMessage != dp_ping_request))
     {
         (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_ERR,
                          DDP_GET_SESSION_STATUS_FAILURE, status,
                         "Can't retrieve status for session
                                  <%ld:%ld> - drop message.",
                          sid.high, sid.low);
         continue;
     }

     if (sstatus == DD_SERVICE_FAILURE_NONEXEC || sstatus ==
                                        DD_SERVICE_FAILURE_EXEC ||
         sstatus == DD_SERVICE_FAILURE_PERMS)
     {
         (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_INFO,
                                   DDP_DROP_MESSAGE, 0,
                                   "Session <%ld:%ld> failed to start - drop
                                                                     message.",
                                   sid.high, sid.low);
              continue;
          }

      /* execute the callback that will process this message */
      switch( ResponseMessage )
       {
                     case dp_connect_confirm:
                            rc = SendConnectConfirmMessage(
                                                        &sid, client_h_p);
                            break;
                     case dp_abort_request:
                            rc = SendAbortRequestMessage(
                                                        &sid, client_h_p);
                            break;
                     case dp_close_request:
                            rc = SendCloseRequestMessage(
                                                        &sid, client_h_p);
                            break;
                     case dp_ping_request:
                            rc = SendPingRequestMessage(&sid, client_h_p);
                            break;
                     case dp_event_confirm:
                            // No confirm needed for this message
                            break;
                     case dp_progress_confirm:
                            // No confirm needed for this message
                            break;
                     case dp_final_stats_confirm:
                            rc = SendFinalStatsConfirmMessage(
                                                        &sid, client_h_p);
                            break;
                     default:
                            EDMDispatch_logent(
                          __FILE__,__LINE__,LOG_ERR,DDP_INVALID_MESSAGE,
                            0,"Invalid message type received.");
       }

    if (rc != 0)
        sleep(1);

    /* Check for a shutdown setting */
  } /* End of forever loop */
 return((void*)0);
} /* End of DispDaemon_ccw() */
```

```
//
// Function: SendConnectConfirmMessage()
// Description:
//      Send the confirm connect message to the
//      dispatch daemon.
//
static int
SendConnectConfirmMessage(
             DD_client_session_id *ssid, rpc_binding_handle_t  *clnt_p )
{
  int *rc = NULL;
  int lrc = 0;
  int savedlrc = 0;
  int status = 0;
  int sstatus = 0;
  DP_connect_confirm_msg *msg_p=NULL;

  if (clnt_p != NULL)
  {
     msg_p = (DP_connect_confirm_msg*)
              calloc(1, sizeof(DP_connect_confirm_msg));
     msg_p->sid.high = ssid->high;
     msg_p->sid.low  = ssid->low;

     rc = dp_connect_confirm_1(msg_p,*clnt_p);

     if (IsDebugOn())
     {
         (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_INFO,
                                 DDP_SENDING_MESSAGE, 0,
                                "Sending dp_connect_confirm_1 message.
                                                                     ");
     }
     free( msg_p );
  }
  else
  {
    rpc_binding_handle_t *client_handle_p = NULL;

    /* Get the csc_binding_handle associated with this sid */
    lrc = GetCSCHandle(ssid,
                       &client_handle_p,
                       &status );

    if (0 != lrc)
    {
        EDMDispatch_logent(
      __FILE__,__LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
                          "GetCSCHandle failed.");
        savedlrc = lrc;
    }

    /* Push message to send onto the queue */
    lrc = PushResponseMessage((int) dp_connect_confirm,
                         *ssid,
                         client_handle_p,
                         &status);
    if (0 != lrc)
    {
        EDMDispatch_logent(
        __FILE__,__LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
                          "PushResponseMessage failed.");
        savedlrc = lrc;
    }
```

```
        lrc = savedlrc;

        return lrc;
    }

    return(0);
}
```

```
//
// Function: SendAbortRequestMessage()
// Description:
//        Send a abort request to a restore service.
//
static int
SendAbortRequestMessage(
            DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
{
  int *rc;
  DP_abort_request_msg *msg_p=NULL;

  msg_p = (DP_abort_request_msg*)
          calloc(1, sizeof(DP_abort_request_msg));
  msg_p->sid.high = ssid->high;
  msg_p->sid.low  = ssid->low;
  rc = dp_abort_request_1(msg_p,*clnt_p);

  if (IsDebugOn())
  {
      (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_INFO,
                                  DDP_SENDING_MESSAGE, 0,
                                 "Sending dp_abort_request_1 message.");
  }

  free( msg_p );
  return(0);
}
```

```
//
// Function: SendCloseRequestMessage()
// Description:
//       Send a close request to a restore service.
//
static int
SendCloseRequestMessage(
            DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
{
  int *rc;
  DP_close_request_msg *msg_p=NULL;

  msg_p = (DP_close_request_msg*)
          calloc(1, sizeof(DP_close_request_msg));
  msg_p->sid.high = ssid->high;
  msg_p->sid.low  = ssid->low;
  rc = dp_close_request_1(msg_p,*clnt_p);

  if (IsDebugOn())
  {
      (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_INFO,
                                 DDP_SENDING_MESSAGE, 0,
                                "Sending dp_close_request_1 message.");
  }

  free( msg_p );
  return(0);
}
```

```
//
// Function: SendPingRequestMessage()
// Description:
//       Send a ping request to a restore service.
//
static int
SendPingRequestMessage(
            DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
{
  int *rc;
  DP_ping_request_msg *msg_p=NULL;

  msg_p = (DP_ping_request_msg*)
          calloc(1, sizeof(DP_ping_request_msg));
  msg_p->sid.high = ssid->high;
  msg_p->sid.low  = ssid->low;
  rc = dp_ping_request_1(msg_p,*clnt_p);

  if (IsDebugOn())
  {
      (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_INFO,
                                 DDP_SENDING_MESSAGE, 0,
                                "Sending dp_ping_request_1 message.");
  }

  free( msg_p );
  return(0);
}
```

```
//
// Function: SendFinalStatsConfirmMessage()
// Description:
//       Send a ping request to a restore service.
//
static int
SendFinalStatsConfirmMessage(
            DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
{
 int status, *rc;
 int out = -1, err = -1;
 int ret = 0;
 rpc_binding_handle_t *ptr;
 DP_final_stats_confirm_msg *msg_p=NULL;

 if (clnt_p != NULL)
 {
     msg_p = (DP_final_stats_confirm_msg*)
             calloc(1, sizeof(DP_final_stats_confirm_msg));
     msg_p->sid.high = ssid->high;
     msg_p->sid.low  = ssid->low;
     rc = dp_final_stats_confirm_1(msg_p,*clnt_p);

     if (IsDebugOn())
     {
         (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_INFO,
                                 DDP_SENDING_MESSAGE, 0,
                                "Sending dp_final_stats_confirm_1
                                                       message.");
     }

     free( msg_p );
 }

 ret = removeSession(ssid, &status);
 if (ret == -1)
  {
     (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_ERR,
                           DDP_REMOVE_SESSION_FAILURE, status,
                          "Failure removing session instance from list.
                                                                ");
  }

 ret = getHandleSet(ssid, &out, &err, &status);
 if (ret == -1)
  {
     (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_ERR,
                           DDP_GET_HANDLE_SET_FAILURE, status,
                          "Failure getting session handles from list.
                                                                ");
  }

 if (out != -1 && err != -1)
 {
    close(out);
    close(err);
 }

 ret = deleteHandleSet(ssid, &ELinkHandle, &status);
 if (ret == -1)
  {
     (void) EDMDispatch_logent( __FILE__,__LINE__, LOG_ERR,
                           DDP_DELETE_HANDLE_SET_FAILURE, status,
                          "Failure removing session handles from list.
                                                                ");
```

```
  }

 return(0);
}
```

```
/*
** ======================================================================
                                                               ========
** Copyright 1996,1997 EMC Corporation
** ======================================================================
                                                               ========
*/
/*
** ======================================================================
                                                               ========
** EDMDD_ccr.cc
**
** Mission Statement: This is the entry point for the Control Channel
                                                                  Reader
**                     thread. Its main purpose is to read asynchronous
**                     messages from the Dispatch Daemon.
**
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                           USE_SUNRPC - Compile source with sunrpc
                                                        support.   If
**                                      not set, assume DCE support.
**
** Basic idea here: Module for Control Channel Reader thread.
** ======================================================================
                                                               ========
*/
/*
** ======================================================================
                                                               ========
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
** ======================================================================
                                                               ========
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMccr.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <signal.h>
#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <pthread.h>
#include <logging/logging.h>
#include <EDMDD_ddp.h>
#include <EDMDD_ccr.h>
#include <csc/cscomm.h>
#include <EDMDispatchLog.h>

static void halt_service(int);
static boolean32 print_error = TRUE;

extern pthread_cond_t cscPortRdy_cv;
extern pthread_mutex_t cscPortRdy_mutex;

void *
```

```
DispDaemon_ccr(void *buff)
{
  int lrc=0;
  error_status_t   status;
  rpc_if_handle_t  if_spec;
  struct esl_timeval timeout = (5,0);

//
// Wait for transient thread to tell me there is somthing to listen on.
//
pthread_mutex_lock( &cscPortRdy_mutex );
pthread_cond_wait( &cscPortRdy_cv, &cscPortRdy_mutex );
pthread_mutex_unlock( &cscPortRdy_mutex );

/*
** ======================================================================
                                                               ========
** Let begin to listen for requests.
** ======================================================================
                                                               ========
*/
  for(;;)
    {
      lrc = csc_async_server_listen( (esl_timeval*)&timeout, &status);
      if ( 1 != lrc )
      {
        EDMDispatch_logent(
                       __FILE__,__LINE__,LOG_ERR,DDP_FAILED_LISTEN,
                0,"Bad returned from listen.");
      }
      if ( 1 == lrc )
      {
        EDMDispatch_logent(
                       __FILE__,__LINE__,LOG_INFO,DDP_LISTEN_TIMEOUT,
                0,"listen() timedout.");
      }
    } /* End of while loop */

/*
** ======================================================================
                                                               ========
**   Unregister our service upon exit request.
** ======================================================================
                                                               ========
*/
  lrc = csc_unregister_async_server_interface(&if_spec, &status);
  EDMDispatch_logent(__FILE__,__LINE__,LOG_INFO,DDP_UNREGISTER_SVC,
                     0,"Returned from unregister service.");
  return((void*)0);
}
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/


/* EDMDDHandle.cc
 *
 *
 * Mission Statement:  file that contains the Handle class methods
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
 *                      The Handle object is a container which holds a
 *                      set of handles for each running service.
 */

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDDHandle.cc,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;

#endif

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <string.h>
#include <stdlib.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>

#include <csc/cscomm.h>

#include <edmlink/edmlink_api.h>
#include <restore/RestoreObjectID.h>
#include <restore/dispatch_daemon.h>
#include <EDMDDHandle.h>

// Needed for rougue wave linked list manager.
// 413 is the object ID.
RWDEFINE_COLLECTABLE(EDMDDHandle, EDMDDHANDLE)

/**********************************************************************
                                                               ********
**
** Routine:  EDMDDHandle constructor
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Initializes the Handle class by resetting the internal
                                                                  data.
**
***********************************************************************
                                                               ********
*/
```

```
EDMDDHandle::EDMDDHandle()
{
    rpcBD = NULL;
    shellHd = NULL;
    stdoutPipe = 0;
    stderrPipe = 0;

    memset(&sessionID, 0, sizeof(sessionID));
}

/**********************************************************************
                                                               ********
**
** Routine:  EDMDDHandle constructor
**
** Inputs:   rpc_binding_handle_t bh - the client handle to use for
                                                                    the
**                                 dispatch protocol
**           int stdoutpipe - the stdout descriptor of the service
**           int stderrpipe - the stderr descriptor of the service
**           DD_client_session_id *sess - the session ID of the
                                                                service
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Initializes the intenals of the Handle class.
**
***********************************************************************
                                                               ********
*/

EDMDDHandle::EDMDDHandle(
          IN rpc_binding_handle_t *bh, IN DD_client_session_id *sess,
             IN int stdoutpipe, IN int stderrpipe)
{
    rpcBD = bh;
    stdoutPipe = stdoutpipe;
    stderrPipe = stderrpipe;

    if (sess != NULL)
       memcpy(&sessionID, sess, sizeof(DD_client_session_id));
}

/**********************************************************************
                                                               ********
**
** Routine:  EDMDDHandle destructor
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Doesn't really do anything but seems to be a requirement
**           for the linked list manager.
**
***********************************************************************
                                                               ********
*/
EDMDDHandle::~EDMDDHandle()
```

```
{
}

/**********************************************************************
                                                               ********

**
** Routine:  compareTo
**
** Inputs:   RWCollectable *c - a pointer to the base class type which
**                              you can then cast and compare.
**
** Outputs:  None
**
** Return Codes:
**           int    -    returns numbers like qsort compare (-1, 0, 1)
**
** Purpose:  Compare using the auxproc PID.
**
**********************************************************************
                                                               ********

*/

int
EDMDDHandle::compareTo(IN const RWCollectable *c) const
{
    EDMDDHandle *localhandle = (EDMDDHandle *) c;

    if (localhandle == NULL)
        return -1;

    if (localhandle -> sessionID.high == sessionID.high &&
        localhandle -> sessionID.low == sessionID.low)
        return 0;
    return (localhandle -> sessionID.high > sessionID.high ||
           (localhandle -> sessionID.high == sessionID.high &&
            localhandle -> sessionID.low > sessionID.low)) ? 1 : -1;
}

/**********************************************************************
                                                               ********

**
** Routine:  isEqual
**
** Inputs:   RWCollectable *c - a pointer to the base class type which
**                              you can then cast and compare.
**
** Outputs:  None
**
** Return Codes:
**           RWBoolean    -    TRUE or FALSE
**
** Purpose:  Compare session IDs to find which session needs service.
**
**********************************************************************
                                                               ********

*/

RWBoolean
EDMDDHandle::isEqual(IN const RWCollectable *c) const
{
    EDMDDHandle *localhandle = (EDMDDHandle *) c;

    if (localhandle == NULL)
        return FALSE;

    if (localhandle -> sessionID.high == sessionID.high &&
```

```
        localhandle -> sessionID.low == sessionID.low)
        return TRUE;
    else
        return FALSE;
}

/**********************************************************************
                                                               ********

**
** Routine:  hash
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           unsigned    -    returns time started
**
** Purpose:  Returns unique value, in this case auxproc pid.
**
**********************************************************************
                                                               ********

*/

unsigned
EDMDDHandle::hash() const
{
    return (unsigned) sessionID.low;
}

/**********************************************************************
                                                               ********

**
** Routine:  saveGuts
**
** Inputs:   RWFile  f   - File pointer where data will be saved.
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Save class internal data to a file.
**
**********************************************************************
                                                               ********

*/

void
EDMDDHandle::saveGuts(IN RWFile &f)
{
    // Save parent class data too
    RWCollectable::saveGuts(f);

    // Left as an example
}

/**********************************************************************
                                                               ********

**
** Routine:  saveGuts
**
** Inputs:   RWvostream strm - stream to write internal data to.
**
** Outputs:  None
**
```

```
** Return Codes:
**          None
**
** Purpose:  Save class data to a stream.
**
*************************************************************************
                                                                ********
*/

void
EDMDDHandle::saveGuts(IN RWvostream &strm)
{
    // Save parent class data too
    RWCollectable::saveGuts(strm);

    // Left as an example
}

/************************************************************************
                                                                ********
**
** Routine:  restoreGuts
**
** Inputs:   RWFile f - file to read internal data from.
**
** Outputs:  None
**
** Return Codes:
**          None
**
** Purpose:  Restores an instance of the Handle class by reading the
                                                                data
**          from the passed in file.
**
*************************************************************************
                                                                ********
*/

void
EDMDDHandle::restoreGuts(IN RWFile &f)
{
    // Restore parent data too
    RWCollectable::restoreGuts(f);

    // Left as an example
}

/************************************************************************
                                                                ********
**
** Routine:  restoreGuts
**
** Inputs:   RWvistream strm - stream to read internal data from.
**
** Outputs:  None
**
** Return Codes:
**          None
**
** Purpose:  Restores an instance of the Handle class by reading the
                                                                data
**          from the passed in stream.
**
*************************************************************************
                                                                ********
*/
```

```
void
EDMDDHandle::restoreGuts(IN RWvistream &strm)
{
    // Restore parent data too
    RWCollectable::restoreGuts(strm);

    // Left as an example
}

/************************************************************************
                                                                ********
**
** Routine:  binaryStoreSize
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**          RWspace count - file size of class written to disk in
                                                                bytes
**
** Purpose:  Returns the size of class if it were stored on disk.
**
*************************************************************************
                                                                ********
*/

RWspace
EDMDDHandle::binaryStoreSize() const
{
    RWspace count = RWCollectable::binaryStoreSize() ;
    return count;
}

/************************************************************************
                                                                ********
**
** Routine:  getSessionID
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**          DD_client_session_id sessionID - the session ID
**
** Purpose:  Returns the ID of the session the object belongs to.
**
*************************************************************************
                                                                ********
*/

DD_client_session_id
EDMDDHandle::getSessionID()
{
    return sessionID;
}

/************************************************************************
                                                                ********
**
** Routine:  setSessionID
**
** Inputs:   DD_client_session_id ID - the session ID associated with
**                                   this object
**
** Outputs:  None
**
** Return Codes:
**              None
**
** Purpose:  Sets the ID of the session the object belongs to.
**
************************************************************************
                                                                ********
*/

void
EDMDDHandle::setSessionID(DD_client_session_id ID)
{
    sessionID = ID;
}

/***********************************************************************
                                                                ********
**
** Routine:  getBindingHandle
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**              rpc_binding_handle_t rpcBD - the binding handle for the
                                                                  client
**                                side of the dispatch protocol
**
** Purpose:  Returns the binding handle for the client side of the
                                                                dispatch
**              protocol.
**
************************************************************************
                                                                ********
*/

rpc_binding_handle_t *
EDMDDHandle::getBindingHandle()
{
    return rpcBD;
}

/***********************************************************************
                                                                ********
**
** Routine:  setBindingHandle
**
** Inputs:   rpc_binding_handle_t *bh - the binding handle to use for
                                                                    this
**                               service
**
** Outputs:  None
**
** Return Codes:
**              None
**
** Purpose:  Sets the binding handle of the session the object belongs
                                                                     to.
**
************************************************************************
                                                                ********
```

```
*/

void
EDMDDHandle::setBindingHandle(rpc_binding_handle_t *bh)
{
    rpcBD = bh;
}

/***********************************************************************
                                                                ********
**
** Routine:  getShellHandle
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**              ELinkShellObjPtr_ty shellHd - the shell handle for the
                                                                  client
**                                side of the dispatch protocol
**
** Purpose:  Returns the shell handle for the client side of the
                                                                dispatch
**              protocol.
**
************************************************************************
                                                                ********
*/

ELinkShellObjPtr_ty *
EDMDDHandle::getShellHandle()
{
    return &shellHd;
}

/***********************************************************************
                                                                ********
**
** Routine:  setShellHandle
**
** Inputs:   ELinkShellObjPtr_ty *bh - the shell handle to use for
                                                                    this
**                               service
**
** Outputs:  None
**
** Return Codes:
**              None
**
** Purpose:  Sets the shell handle of the session the object belongs
                                                                     to.
**
************************************************************************
                                                                ********
*/

void
EDMDDHandle::setShellHandle(ELinkShellObjPtr_ty *bh)
{
    shellHd = *bh;
}

/***********************************************************************
                                                                ********
**
```

```
** Routine:  getStdoutPipe
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           int stdoutFD - the stdout descriptor of the service
**
** Purpose:  Returns the stdout handle of the service.
**
************************************************************************
                                                                ********
*/

int
EDMDDHandle::getStdoutPipe()
{
    return stdoutPipe;
}

/***********************************************************************
                                                                ********
**
** Routine:  setStdoutPipe
**
** Inputs:   int handle - the stdout handle of the private service
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Sets the stdout handle of the private service.
**
************************************************************************
                                                                ********
*/

void
EDMDDHandle::setStdoutPipe(int handle)
{
    stdoutPipe = handle;
}

/***********************************************************************
                                                                ********
**
** Routine:  getStderrPipe
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           int stderrPipe - the stderr descriptor of the service
**
** Purpose:  Returns the stderr descriptor of the service.
**
************************************************************************
                                                                ********
*/

int
EDMDDHandle::getStderrPipe()
{
```

```
    return stderrPipe;
}

/***********************************************************************
                                                                ********
**
** Routine:  setStderrPipe
**
** Inputs:   int handle - the stderr handle of the service
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Sets the stderr handle of the service.
**
************************************************************************
                                                                ********
*/

void
EDMDDHandle::setStderrPipe(int handle)
{
    stderrPipe = handle;
}
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/


/* EDMDDHandleMgrApi.cc
 *
 *
 * Mission Statement:  An API to manage the handle sets/objects
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
 *                      This API manages the handle sets.
                                                 Multiple threads
 *                     need access to the handles to do IO.
 *
 *                     Each time an fd_set is modified or used we lock
 *                     a mutex to make sure access is serialized.
 */

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDDHandleMgrApi.cc,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <pthread.h>

#include <EDMDDHandleMgrApi.h>

// The tree that we keep handles in. We could have used any
// Rogue Wave object but I decided to use the tree.

static RWBinaryTree G_handleTree;

// These are the fd_sets managed on behalf of the user. The
// Modified sets are changed any time new handles are added
// and the others are the actual copies given to the user.
// This allows the manager to add handles from 1 thread without
// directly effecting a select call made in another thread.

fd_set   stdoutSetModified,
         stdoutSet,
         stderrSetModified,
         stderrSet;

// These are values for the highest number handle that is part of a
// given set. Keep in mind that 1 has to be added to this number to
// select on the highest handle used.

int highStdout = 0,
    highStderr = 0;

static pthread_mutex_t    G_fdSetMtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t    G_handleTreeMtx = PTHREAD_MUTEX_INITIALIZER;
```

```
/*********************************************************************
                                                              ********
**
** Routine:  initFdSets
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Initialize fd sets and mutex.
**
*********************************************************************
                                                              ********
*/

static void
initFdSets()
{
   FD_ZERO(&stdoutSetModified);
   FD_ZERO(&stdoutSet);
   FD_ZERO(&stderrSetModified);
   FD_ZERO(&stderrSet);

   pthread_mutex_init(&G_fdSetMtx, NULL);
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  LockHandleMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the mutex for the handle tree object
**
************************************************************************
                                                                ********
*/

static void
LockHandleMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_handleTreeMtx, NULL);
    }

    pthread_mutex_lock(&G_handleTreeMtx);
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  UnlockHandleMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for the handle tree object
**
************************************************************************
                                                                ********
*/

static void
UnlockHandleMutex()
{
    pthread_mutex_unlock(&G_handleTreeMtx);
}
```

```
/**************************************************************************
                                                                  ********
**
** Routine:  getStdoutSet
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           fd_set * - the stdoutSet...
**
** Purpose:  Returns the stdoutSet fd_set after the stdoutSetModified
                                                                       was
**           copied into it. Modified is the most recent copy.
**
**************************************************************************
                                                                  ********
*/

int
getStdoutSet(fd_set *yourset, int *highhandle, int *status)
{
    if (status == NULL)
    {
        return -1;
    }

    if (yourset == NULL || highhandle == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    pthread_mutex_lock(&G_fdSetMtx);

    stdoutSet = stdoutSetModified;

    memcpy(yourset, &stdoutSet, sizeof(fd_set));

    *highhandle = highStdout;

    pthread_mutex_unlock(&G_fdSetMtx);

    return 0;
}
```

```
/**************************************************************************
                                                                  ********
**
** Routine:  getStderrSet
**
** Inputs:   None
**
** Outputs:  fd_set *yourset - the fd_set for the from descriptors
**           int    *highhandle - the highest handle for this set
**           int    *status - there's no status to return right now
**                            but leave it as a placeholder
**
** Return Codes:
**           int - 0 for success or -1 for failure.
**
** Purpose:  Returns the stderrSet fd_set after the stderrSetModified
                                                                       was
**           copied into it. Modified is the most recent copy.
**
**************************************************************************
                                                                  ********
*/

int
getStderrSet(fd_set *yourset, int *highhandle, int *status)
{
    if (status == NULL)
    {
        return -1;
    }

    if (yourset == NULL || highhandle == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    pthread_mutex_lock(&G_fdSetMtx);

    stderrSet = stderrSetModified;

    memcpy(yourset, &stderrSet, sizeof(fd_set));

    *highhandle = highStderr;

    pthread_mutex_unlock(&G_fdSetMtx);

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  LookupHandleSet
**
** Inputs:   DD_client_session_id *sess - the session ID to lookup
                                                                  with
**
** Outputs:  int *status - a place to put a status if something goes
                                                               wrong.
**           EDMDDHandle **hs - the handle set to return
**
** Return Codes:
**           0 for success and non-zero otherwise
**
** Purpose:  Looks up a handle set.
**
************************************************************************
                                                                ********
*/

int
LookupHandleSet(
          DD_client_session_id *sess, EDMDDHandle **hs, int *status)
{
    EDMDDHandle  *ret, *handleobj;

    if (status == NULL)
    {
        return -1;
    }

    if (hs == NULL || sess == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    handleobj = new EDMDDHandle();

    if (handleobj == NULL)
    {
        *status = HANDLEMGR_NO_MEMORY;
        return -1;
    }

    handleobj -> setSessionID(*sess);

    LockHandleMutex();

    ret = (EDMDDHandle *) G_handleTree.find(handleobj);

    UnlockHandleMutex();

    delete handleobj;

    if (ret == NULL)
    {
        *status = HANDLEMGR_LOOKUP_FAILED;
        return -1;
    }

    *hs = ret;

    return 0;
```

```
}
```

```
/**********************************************************************
                                                               ********
**
** Routine:  newHandleSet
**
** Inputs:   int stdouthandle - the handle to send commands to auxproc
**           int stderrhandle - the handle to receive responses from
                                                                auxproc
**           rpc_binding_handle_t *connhandle - the connection handle
**           ELinkShellObjPtr_ty *shell - the shell handle
**
** Outputs:  int *status - a place to put a status if something goes
                                                                 wrong.
**
** Return Codes:
**             0 for success and non-zero otherwise
**
** Purpose:  Creates a new handle set.
**
***********************************************************************
                                                               ********

*/

int
newHandleSet(
        DD_client_session_id *sess, int stdouthandle, int stderrhandle,
                rpc_binding_handle_t *connhandle,
                                        ELinkShellObjPtr_ty *shell,
                int *status)
{
    EDMDDHandle *handle;
    EDMDDHandle *ret;
    int          flags = 0;
    static boolean_ty  first = TRUE;

    if (first == TRUE)
    {
        initFdSets();
        first = FALSE;
    }

    if (status == NULL)
    {
        return -1;
    }

    if (sess == NULL || connhandle == NULL || shell == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    handle = new EDMDDHandle();

    if (handle == NULL)
    {
        *status = HANDLEMGR_NO_MEMORY;
        return -1;
    }

    handle -> setStdoutPipe(stdouthandle);
    handle -> setStderrPipe(stderrhandle);
    handle -> setBindingHandle(connhandle);
    handle -> setShellHandle(shell);
    handle -> setSessionID(*sess);
```

```
    pthread_mutex_lock(&G_fdSetMtx);

    flags = fcntl(stdouthandle, F_GETFL, 0);
    fcntl(stdouthandle, F_SETFL, flags | O_NDELAY);

    flags = fcntl(stderrhandle, F_GETFL, 0);
    fcntl(stderrhandle, F_SETFL, flags | O_NDELAY);

    FD_SET(stdouthandle, &stdoutSetModified);
    FD_SET(stderrhandle, &stderrSetModified);

    if (stdouthandle > highStdout)
    {
        highStdout = stdouthandle;
    }

    if (stderrhandle > highStderr)
    {
        highStderr = stderrhandle;
    }

    pthread_mutex_unlock(&G_fdSetMtx);

    LockHandleMutex();

    ret = (EDMDDHandle *) G_handleTree.insert(handle);

    UnlockHandleMutex();

    if (ret == NULL)
    {
        *status = HANDLEMGR_INSERT_FAILED;
        delete handle;
        return -1;
    }

    return 0;
}
```

```
/*********************************************************************
                                                             ********
**
** Routine:  getHandleSet
**
** Inputs:   DD_client_session_id *sess - a session ID to use to look
                                                                   up
**                                         the handle set
**
** Outputs:  int *status - a place to put a status if something goes
                                                               wrong.
**           int *sout - stdout descriptor for the service.
**           int *serr - stderr descriptor for the service.
**
** Return Codes:
**           0 for success and non-zero otherwise
**
** Purpose:  Removes a handle set.
**
*********************************************************************
                                                             ********
*/

int
getHandleSet(
         IN DD_client_session_id *sess, OUT int *sout, OUT int *serr,
            OUT int *status)
{
    EDMDDHandle *handle;
    int          lret;

    if (status == NULL)
    {
        return -1;
    }

    if (sess == NULL || sout == NULL || serr == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    lret = LookupHandleSet(sess, &handle, status);

    if (lret != 0)
        return lret;
    else if (handle == NULL)
        return -1;

    *sout = handle -> getStdoutPipe();
    *serr = handle -> getStderrPipe();

    return 0;
}
```

```
/*********************************************************************
                                                             ********
**
** Routine:  GetCSCHandle
**
** Inputs:   DD_client_session_id *sess - a session ID to use to look
                                                                   up
**                                         the handle set
**
** Outputs:  int *status - a place to put a status if something goes
                                                               wrong.
**           rpc_binding_handle_t *cscb - binding handle for this
                                                              session
**
** Return Codes:
**           0 for success and non-zero otherwise
**
** Purpose:  Returns CSC binding handle.
**
*********************************************************************
                                                             ********
*/

int
GetCSCHandle(
        IN DD_client_session_id *sess, OUT rpc_binding_handle_t **cscb,
           OUT int *status)
{
    EDMDDHandle *handle;
    int          lret;

    if (sess == NULL || cscb == NULL || status == NULL)
    {
        return -1;
    }

    lret = LookupHandleSet(sess, &handle, status);

    if (lret != 0)
        return lret;
    else if (handle == NULL)
        return -1;

    *cscb = handle -> getBindingHandle();

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  GetShellHandle
**
** Inputs:   DD_client_session_id *sess - a session ID to use to look
                                                                      up
**                                  the handle set
**
** Outputs:  int *status - a place to put a status if something goes
                                                                  wrong.
**           ELinkShellObjPtr_ty **shell - shell handle for this
                                                                 session
**
** Return Codes:
**              0 for success and non-zero otherwise
**
** Purpose:  Returns shell handle.
**
************************************************************************
                                                                ********
*/

int
GetShellHandle(
       IN DD_client_session_id *sess, OUT ELinkShellObjPtr_ty **shell,
             OUT int *status)
{
    EDMDDHandle *handle;
    int         lret;

    if (sess == NULL || shell == NULL || status == NULL)
    {
        return -1;
    }

    lret = LookupHandleSet(sess, &handle, status);

    if (lret != 0)
        return lret;
    else if (handle == NULL)
        return -1;

    *shell = handle -> getShellHandle();

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  deleteHandleSet
**
** Inputs:   DD_client_session_id *sess - a session ID to use to look
                                                                      up
**                                  the handle set
**
** Outputs:  int *status - a place to put a status if something goes
                                                                  wrong.
**
** Return Codes:
**              0 for success and non-zero otherwise
**
** Purpose:  Removes a handle set.
**
************************************************************************
                                                                ********
*/

int
deleteHandleSet(DD_client_session_id *sess, ELinkHandlePtr_ty *hand,
                int *status)
{
    EDMDDHandle *handle;
    EDMDDHandle *ret;
    int          lret;
    rpc_binding_handle_t *bh;
    ELinkShellObjPtr_ty *shell;
    error_status_t err;

    if (status == NULL)
    {
        return -1;
    }

    if (sess == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    lret = LookupHandleSet(sess, &handle, status);

    if (lret != 0)
        return lret;
    else if (handle == NULL)
        return -1;

    LockHandleMutex();

    ret = (EDMDDHandle *) G_handleTree.remove(handle);

    UnlockHandleMutex();

    if (ret == NULL)
    {
        *status = HANDLEMGR_REMOVE_FAILED;
        delete handle;
        return -1;
    }

    pthread_mutex_lock(&G_fdSetMtx);
    FD_CLR(ret -> getStdoutPipe(), &stdoutSetModified);
    FD_CLR(ret -> getStderrPipe(), &stderrSetModified);

    pthread_mutex_unlock(&G_fdSetMtx);

    bh = ret -> getBindingHandle();

    csc_free_binding(bh, 0, &err);

    shell = ret -> getShellHandle();

    ELinkDestroyObj(hand, *shell);

    delete ret;

    return 0;
}
```